MP3 decoding on FPGA: a case study for floating point acceleration

Alexandros Papakonstantinou, Yohannes Kifle, Gregory Lucas, Deming Chen
ECE Department, University of Illinois, Urbana - Champaign
{apapako2, ykifle2, gmlucas2, dchen} @uiuc.edu

Abstract

Reconfigurable devices are becoming an extremely attractive means for computing and prototyping. The main reasons for their popularity are fast turnaround time, cheap implementation (low/medium volume), easier design flow (no physical implementation or fabrication involvement), and great speed-up potential through application parallelism mapping on the reconfigurable fabric. Moreover, with embedded hard core and soft core processors, reconfigurable devices provide an excellent framework for hardware-software co-design with even shorter total turn-around time. This work constitutes a case study for the acceleration of Floating Point (FP) applications, using the NIOS2 processor. Our case study is based on a software MP3 decoder implementation that heavily relies on floating point math. NIOS2 is a simple microprocessor that has no integrated floating point (FP) units; thus, a set of FP accelerators is necessary for achieving real-time or faster decoding. We explore two main issues: i) the speedup and efficiency of FP accelerators on the FPGA; and ii) the effect of communication overhead between the processor and the FP accelerators. We show that real-time decoding is possible by leveraging a) simple software modifications, b) efficient accelerator designs, and c) smart parallelism extraction strategies.

1. Introduction

Field Programmable Gate Arrays (FPGAs) have advanced to a high level of sophistication and logic capability during the past decade. In addition, the corresponding design, verification and optimization tools have also reached high maturity and capacity. Thus, although once considered mainly for the implementation of glue logic or generation of prototypes for digital systems, now FPGAs have become a major source of computation horsepower in different types of applications. With embedded soft-core and hard-core processors, on chip memory, register files and other peripherals, FPGAs are being employed as stand-alone SoCs (System-on-a-Chip) or powerful accelerators in large multi-chip systems.

Apart from the great design flexibility inherent in their reconfigurability and the potential for massive parallelism exploitation on their reconfigurable fabric, FPGAs offer additional important benefits. Specifically, they provide fast total turnaround time, cheap implementation (at low/medium volume) and simpler design flows (no physical implementation or fabrication involvement). These features are of paramount importance as the fabrication mask costs for ASIC designs are sky-rocketing in the deep submicron technologies, and time-to-market is becoming a critical survival metric in the semiconductor industry.

In this work we leverage these FPGA benefits to efficiently implement an MPEG layer-3 (MP3) audio decoder that relies heavily on floating point (FP) operations. We port a floating point C implementation onto NIOS2, Altera's soft-core processor and use it as our baseline. In order to achieve real-time decoding for our MP3 player we experiment with different FP acceleration schemes. Our optimization strategy is guided by profile information for the MP3 decoder. During our evaluation of the different options we also consider the effect of communication overhead on performance between NIOS2 and the accelerators,. We show that real-time and even faster decoding can be achieved in a non-complicated methodology that involves simple software modifications, small accelerator designs, and smart communication and parallelism extraction strategies.

In the following section we briefly describe some previous work carried out in this area. In section 3 a description of the hardware and software platform that we use is given, followed by descriptions of our acceleration strategies in section 4. Section 5 presents the results of our different acceleration techniques and section 6 concludes this paper.

2. Related work

Audio coding and decoding has been a very popular application for custom acceleration implementations. The authors in [1] and [2] designed custom acceleration units for critical computation functions of MP3 decoding and encoding respectively. They built their accelerators for the AMBA bus, and use them within ARM-based systems. A similar methodology is used for the Ogg Vorbis decoder acceleration in [4]. On a slightly different perspective, the work in [3] proposes a stand alone implementation which is based on two types of custom cores, a RISC-like core for the decision making and dedicated acceleration engines for specific computation parts. MP3 decoding is used as a representative application for automated SoC design through the use of SpecC for design exploration and hardware-software partitioning in [5]. All of these works have avoided using floating point hardware by converting the math operations into fixed point. In this work we used a floating point implementation and implemented our accelerators as custom instructions, rather than custom peripherals.

3. Design Environment

We base our work on Altera's DE2 board which hosts a Cyclone2 FPGA, SDRAM, SRAM and FLASH memories,

audio and VGA codecs, switches, LEDs and other components that offer great flexibility for implementing a wide range of applications. For our work we used Cyclone2, the SDRAM memory (8MBs) for code and data storage, both oscillators of the board and the audio codec (WM8731) for converting the audio samples into analog signals. Cyclone2 is at the low-end of FPGAs in terms of capacity and capabilities, but this did not pose any problems for the acceleration of the MP3 decoder. In particular, Cyclone is a 33K logic-element device with embedded RAM blocks (105), multipliers (35) and PLLs (4). For the implementation, debugging and evaluation of our designs we used several CAD tools: Quartus II for compilation and synthesis, the SOPC Builder for generating Avalon-bus based NIOS2 systems, the NIOS2 IDE for software development and Modelsim for simulation.

The main two constituents that we used as the baseline for the MP3 decoder are NIOS2 processor and a floating-point implementation of MP3 decoding in C by M. Ruckert [6]. NIOS2 is a relatively simple, in-order soft-core processor that can be easily customized and connected with various peripherals on the Avalon bus through the SOPC Builder tool. However, NIOS2 does not include any FP units in its basic architecture. Thus, computation of FP operations on NIOS2 is very slow and inefficient as software libraries are used instead. Nonetheless, NIOS2 can make use of some hardware floating-point extensions that can be added to the basic architecture and help speed up single-precision FP (SPFP) operations in a transparent way (no change in the C code is required). On top of these FP extensions we design some new FP accelerators that are attached to NIOS2 through its custom instruction extensibility feature. That is, NIOS2 can accommodate new custom instructions that are associated with custom hardware that takes two 32-bit inputs and produces a 32-bit result[1].

The FP implementation of MP3 (mp32pcm [6]) we used in our baseline system is a highly efficient software decoder that was written for sophisticated processors used in user PCs. These processors use sophisticated FPUs and can provide high FP computation power to the applications. Using mp32pcm, an mp3 file can be decoded in less than 5 seconds on these processors. The original code of mp32pcm uses double-precision (DPFP) math, but we converted all double-precision to single-precision in order to use the transparent SPFP extensions for our initial evaluations. Subsequently, we also used the original DPFP application to carry out a preliminary exploration on DPFP acceleration.

3.1. MP3 decoding system overview

Fig. 1 gives an overview of the system setup we used. NIOS2 runs at 83.3MHz and has a 32KB data cache and a 4KB instruction cache. It is also tightly connected to the custom instruction (CI) unit. Through the Avalon bus it communicates with a FIFO where it stores the decoded audio samples. At the other end of the FIFO an audio controller reads the audio samples and sends them to the audio codec that sits on the DE2 board.

[1] Through supporting this feature, NIOS2 falls into the domain of ASIP architecture (application-specific instruction set processor).

4. Acceleration schemes

In our acceleration experiments, the first step was to use the NIOS2 FP extensions which are implemented as custom instructions and can perform SPFP addition and multiplication and optionally division (was not used in our setup). This had a big impact in the MP3 decoding performance, since all the FP operations were dispatched to the custom instruction hardware and the use of software libraries for FP math was eliminated. A big advantage of using the FP extensions of NIOS2 is that no modification of the C code is required, as the C compiler automatically transforms all the FP operations to custom instruction calls. On the downside, the user has no control over the characteristics of these FP extensions and therefore the performance benefits may not be optimal for all applications.

The second step was to profile the execution of the MP3 decoder on the FP-extended NIOS2. By using the profiling tool provided in the NIOS2 IDE, we discovered that most of the execution time is spent on three functions: i) *windowing* (59%), ii) *dct32* (15%) and iii) *dct18* (8%). These functions account for 82% of the total execution time and they all comprise of FP additions and multiplications. The windowing function computes 32 samples by using a highly regular pattern of multiply-add pairs. On the contrary, the dct functions use irregular patterns of FP operations, making it harder to find common patterns that can share a common accelerator. By taking into consideration the fact that windowing is by far the most time-consuming function of the MP3 decoder, we decided to focus our acceleration efforts mainly on this function.

4.1. Accelerator designs

In order to accelerate the windowing function we used different configurations of multiply-accumulate (MAC) hardware. Our basic MAC (fig. 2a) accelerator comprises of a FP multiplier and a FP Adder/Subtractor. Both modules are built using the Quartus-II MegaFunction library and they are not pipelined (pipelining is not available for MegaFunction FP modules). In order to improve the throughput, registers are added between the multiplier and the adder, so that two MAC operations can be pipelined in the MAC accelerator. A

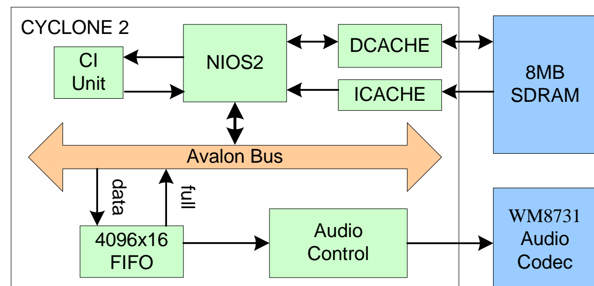
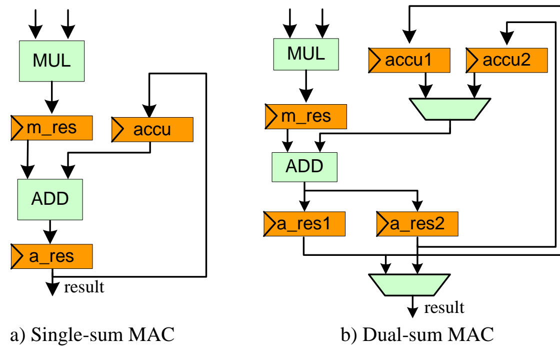


Fig 1. MP3 decoder system overview

DPFP version of this MAC was implemented by extending the datapath to 64bits. The DPFP MAC unit is also used for simple DPFP multiplications and additions, since there are no ready DPFP CI extensions. Fig. 2b shows an enhanced MAC accelerator that can be used for the computation of two independent sums. In this design the original MAC is simply extended by some extra registers and MUXes and can be easily scaled for computing more than two independent sums (multi-sum MAC). Each sum is accumulated in a separate register, and an input signal selects which sum the input operands refer to. Our MAC accelerators are implemented as NIOS2 custom instructions. In the case of the DPFP MAC and the multi-sum MAC, the different operations and the different sums respectively, are identified by a special field in the instruction word which is not decoded by NIOS2 decoder. Instead it is extracted from the instruction word and sent to the custom instruction module. The accelerator uses this field to guide the operands and results to the right registers and datapath units.

4.2. SW modifications for parallelism extraction

The windowing function computes 32 sums that are independent from each other (i.e. none of the sums depends on the previously calculated sums). Each sum comprises of 16 accumulated multiplications. Every sum is calculated by calling the MAC custom instruction 16 times. The benefit of using the MAC unit instead of a separate adder and multiplier is that NIOS2 does not need to store the intermediate results of accumulations, thus saving considerable time on register-file and memory accesses. In order to achieve the improved throughput in the pipelined MAC accelerator, we had to set up NIOS2 so that it perceives the latency of the custom instruction as equal to the latency of the MAC pipeline stage (i.e. half of the full MAC latency). This results in every MAC operation returning the partial sum up to the previous accumulation. The reduced latency works fine in the case of the first n-1 accumulations of a sum, since the partial sum is not used by the application. However, in the case of the nth accumulation we need to ensure that the CI returns the correct sum. This is achieved by adding a fake MAC operation (with zero operand values) after the final accumulation operation of the sum. The fake MAC operation does not alter the sum but its purpose is to



a) Single-sum MAC b) Dual-sum MAC

Fig 2. MAC Configurations

allow the previous accumulation to proceed to the end of the MAC pipeline and to return the final sum result.

In order to exploit more parallelism on NIOS2 we can reduce the perceived latency of the CIs even more by using more MAC units and interleaving accumulations from independent sums. For example, by interleaving the accumulations of two independent sums on two MAC accelerators we can reduce the perceived latency to as low as 1 cycle as we will see in section 5.

The main disadvantage of the custom instruction acceleration technique is that NIOS2 has to call a MAC CI for every operand pair of an accumulation, which restricts the amount of speedup. This has a dramatic effect in performance in the case of DPFP CIs since NIOS2 has a 32-bit CI interface. This results in extra accesses for sending the operands to and receiving the result from the DPFP CI. Alternatively, a DMA controller could be used to feed the MAC with the sequence of operand pairs, thus freeing NIOS2 to work on the sequential part of the MP3 decoder. However, implementing this in the windowing function would require significant software modifications so as to coalesce the accumulated operands in memory, which is beyond the scope of this work.

5. Experimental Results

Table 1 lists the speedups achieved for the windowing and the total decoding execution by our different SPFP MAC configurations. Both speedups over basic NIOS2 and over FP-extended NIOS2 are shown. The resource usage on the Cyclone2 FPGA is shown in table 2 for each configuration (the numbers in parentheses show the resource overhead over the FP-extended NIOS2).

Initially we used a single MAC unit (Fig 2a) to accelerate the windowing function. A speedup of 1.19X over the FP-extended NIOS2 for the windowing function was observed, which was translated into a 1.14X speedup of the mp3 decoding. The corresponding speedup over the basic NIOS2 without any FP extensions (Speedup vs. noFP) is 26.55 and 22.88 respectively. The MAC custom instruction was set to 14 cycles in NIOS2 for this case.

Subsequently, two separate MAC units were attached to NIOS2, and they were associated to two different custom instructions. Each MAC was used to calculate 16 out of the 32 windowing sums and the perceived latency of both custom instructions was reduced to 7 cycles. Each pair of sums was calculated concurrently by executing the accumulations of the two sums in an interleaved fashion. This is possible since no dependence exist between the windowing sums. In this way, speedups of 1.21X and 1.15X for windowing and decoding were achieved respectively (3rd row). This scheme does not make use of the pipeline in the MACs, but achieves similar throughput through interleaving. Moreover, it requires the appending of fake MAC instructions at the end of sum calculations.

For the previous dual-MAC scheme, we measured the average latency of each MAC custom instruction call and found it to be more than 14 cycles. This latency includes the time spent on loading the input operands from memory or the

register file and any other operations NIOS2 has to complete before calling a custom instruction. This led us to try to speed up the previous scheme by assigning single cycle latency to both MAC custom instructions. The experiment was successful and our speedups rose to 1.30X and 1.19X for windowing and decoding respectively (4th row). This optimization helped us also meet real-time decoding requirements.

Subsequently we increased the number of MACs to four to see if any further performance benefit could be gained by the extra compute power. Similar to the dual-MAC case, the accumulations of 4 sums in windowing were interleaved. Each MAC was set up with one cycle latency. As can be seen in the result table, further speedup was achieved: 1.33X and 1.21X. In order to reduce the acceleration hardware without impacting performance, we tried implementing the quad-MAC scheme with a dual-MAC configuration, using two modified MACs (fig. 2b), each of which can store two sums. Four sums are calculated on the two MACs in an interleaved fashion:

M1-S1, M2-S2, M1-S3, M2-S4

where M1 and M2 are the two modified MACs and S1-S4 are the four sums. We can see in table 1 that performance drops a little bit compared to the 4xMAC case. This is probably due to extra operations that NIOS2 has to execute to handle more CIs.

Table 1: SPFP Speedup results

	Windowing		Decoding	
Custom Instr.	Speedup vs. noFP	Speedup vs. FP-ext	Speedup vs. noFP	Speedup vs. FP-ext
1-MAC 14-cycle	26.55	1.19	22.88	1.14
2xMAC 7-cycle	26.93	1.21	22.94	1.15
2xMAC 1-cycle	28.94	1.30	23.86	1.19
4xMAC 1-cycle	29.72	1.33	24.22	1.21
2x2MAC 1-cycle	28.21	1.27	23.55	1.18

Table 2: SPFP Resource usage

Custom Instr.	Logic Element number	Embedded Multipliers
1-MAC	9292 (+1213)	18 (+7)
2xMAC	10258 (+2179)	25 (+14)
4xMAC	12311 (+4232)	39 (+28)
2x2MAC	10498 (+2419)	25 (+14)

Finally we carried out a preliminary study on the possible acceleration of the DPFP version of our application with the CI technique. A difference from the SPFP experiments was that all FP operations had to use our CI unit, since no DPFP extensions exist in NIOS2. This required modifying the C code to call the DPFP CI for every FP multiplication, addition and MAC operation. Due to the 32-bit width of the CI interface, extra overhead was added by operand splitting into 32bit values, merging them back to 64bit DP format, and the additional required calls of the CI function. Two of the consequences of this overhead are that i)a large fraction of the acceleration is lost in the communication between NIOS2 and the DPFP CI unit and ii)the addition of more MAC units does not offer but negligible performance improvement. Table 3 compares the performance and hardware resources of the DPFP and SPFP acceleration schemes using a single MAC over the purely software implementation (noFP). Note that the SPFP configuration contains the SPFP extension as well as the SPFP MAC unit, while in the DPFP configuration the DPFP MAC is the only CI module.

Table 3: DPFP & SPFP vs. noFP

FP Scheme	Speedup		Extra HW	
	Windowing	Decoding	Logic Elem.	Multipliers
DPFP	13.37	7.31	+3242	+18
SPFP	30.06	26.77	+3311	+14

6. Conclusions

We have used a fairly simple microprocessor and designed simple but efficient accelerators to achieve real-time and faster decoding of MP3 data. By pipelining our accelerators and interleaving independent calculations in the windowing function we managed to extract parallelism out of the application and map it on a simple in-order processor without using stand-alone accelerators and DMA techniques. We demonstrated that with a few simple code modifications and efficient custom instruction extensions we were able to quickly build an SPFP-based MP3-player on an FPGA for real time decoding. We also presented preliminary results for DPFP acceleration. Since NIOS2 CI interface is 32-bit wide, DPFP CIs are not as efficient. This is due to the extra communication and the data leveraging costs.

7. References

[1] S. Gadd and T. Lenart, A hardware accelerated mp3 decoder with Bluetooth streaming capabilities, Master's thesis, Lund Institute of Technology, Sweden, 2001.

[2] Y. Lu, C. Shen and C. Chen, A novel hardware accelerator architecture for MPEG-2/4 AAC encoder, *ICME 2004*

[3] T. Tsai, L. Chen and R. Wu, A Cost-Effective Design for MPEG-2 Audio Decoder with Embedded RISC Microprocessor, *SIPS99 1999*

[4] A. Kosaka, et al., A Hardware Implementation of Ogg Vorbis Audio Decoder with Embedded Processor, *ITCCSCC 2002.*

[5] P. Chandraiah and R. Domer, Specification and Design of a MP3 Audio Decoder, Technical Report CECS-05-04, University of California, Irvine 2005.

[6] M. Ruckert, *Understanding MP3: syntax, semantics, mathematics, and algorithms*, Vieveg 2005.